**React**

**Props and State**

props (short for “properties”) and state are both plain JavaScript objects. While both hold information that influences the output of render, they are different in one important way: props get passed to the component (similar to function parameters) whereas state is managed within the component (similar to variables declared within a function).

**State**

* Role of the component is to translate raw data into HTML, we can consider that this raw data is made out of props and state objects. We could even say that props and state are the input data for the render() function. So the State represents some property (or properties) that controls the behavior of the component.
* State is a plain JavaScript object used by React to represent an information about the component’s current situation.
* It’s managed in the component (just like any variable declared in a function). The difference is while a “normal” variable “disappears” when their function exits, the state variables are preserved by React.
* React has another special built-in object called state, which allows components to create and manage their own data. So unlike props, components cannot pass data with state, but they can create and manage it internally.
* “**Normal**” variable example shows below
* const Home = () => {
* //the value is not reactive without state
* let name = 'Bhushan';
* const handleName = () => {
* name = 'Kumar';
* }
* return(
* <div className="square border">
* <h1>Using variable</h1>
* <div className="d-flex flex-row mb-3">
* <div className="p-2"><h1>Home</h1></div>
* <div className="p-2"><h1>{name}</h1></div>
* <div className="p-2"><button onClick={handleName}>Change Name</button></div>
* </div>
* </div>
* );
* }
* export default Home;

Here if we try to change the name it will not reflect.

* **State** variable example shows below
* import React from "react";
* class Contact extends React.Component {
* constructor(){
* super();
* this.state ={
* id: 1,
* name: "test",
* }
* }
* updateName = () => {
* this.setState({ name: "Simplilearn" });
* }
* render() {
* return(
* <div>
* {this.state.name}
* <button onClick={this.updateName}>Click</button>
* </div>
* )
* }
* }
* export default Contact;

Here the state value is reactive.

**State:** The state is a built-in React object that is used to contain data or information about the component. For initialize state inside constructor use(this.state = { propertyName : Value})

**Constructor:** The constructor is a method used to initialize an object's state in a class. The constructor in a React component is called before the component is mounted.

**Super():** The super() is used inside the constructor for the purpose to get access to this keyword inside our constructor.

Super is a keyword in JavaScript and is used to call super or parent class in the hierarchy. React class extends React Component with ES6 syntax. If we would like to set a property or access this inside the constructor we need to call super() method.

**updateName():** It’s a function to call while user click on button. Here we update the state data using setState().

**SetState():** we use the setState() method to change the state object.

State life cycle any?

* + your state is lost when the page is refreshed (i.e. when the component is unmounted)
  + when you navigate to another component/page, your state is NOT lost. With react router navigations and react re-renders, the state is not lost.
  + If you want to retain state after page refresh then you need to store your state in localStorage & retain the state upon component mount.

**Props**

* Props stand for "**Properties**." They are **read-only** components.
* It is an object which stores the value of attributes of a tag and work similar to the HTML attributes.
* It gives a way to pass data from one component to other components.
* It is similar to function arguments. Props are passed to the component in the same way as arguments passed in a function.

Calling component

* <PropsExample name='testProps' id='10' />

Child component

* import React from 'react';
* class PropsExample extends React.Component{
* constructor(props){
* super(props);
* this.state ={
* id: 1,
* name: "test"
* }
* }
* render(){
* return(
* <div className="square border">
* <p>This is props example: {this.props.name}, {this.props.id}</p>
* </div>
* );
* }
* }
* export default PropsExample;

**{this.props.children}**

If we want to send any data from parent to children must be enclosed within opening and closing of the parent component tag. Shown below.

<PropsExample name='testProps' id='10'>Children data</PropsExample>

Then we can receive it in children component using props.children. Shown below.

import React from 'react';

class PropsExample extends React.Component{

    constructor(props){

        super(props);

    }

    // changeProps = () => {

    //     this.props.id = 10;

    // }

    render(){

        return(

            <div className="square border">

            <p>{this.props.children}</p>

            </div>

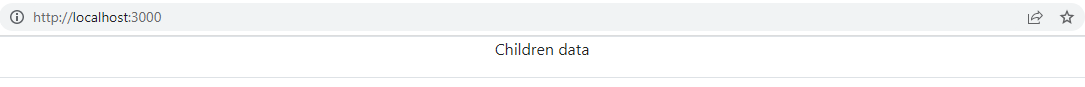
        );

    }

}

export default PropsExample;

Output is,



**PropTypes**

We can use React props, short for properties, to send data from one component to another. If a component receives the wrong type of props, it can cause bugs and unexpected errors in your app.

* React has an internal mechanism for props validation called PropTypes.
* When developing a React application, you’ll need to structure and define your props to avoid bugs and errors.
* Just like a function might have mandatory arguments, a React component might require a prop to be defined, otherwise, it will not render properly.
* Forgetting to pass a required prop into a component that needs it could cause your app to behave unexpectedly.
* When props are passed to a React component, they are checked against the type definitions configured in the propTypes property. When an invalid value is passed for a prop, a warning is displayed on the JavaScript console.
* If default props are set for the React component, the values are first resolved before type checking against propTypes. Therefore, default values are also subject to the prop type definitions.
* Keep in mind that type checking propTypes can happen only in development mode, enabling you to catch bugs in your React application before releasing it to the production environment.
* Install it by running this command npm install prop-types --save
* Import it to component import PropTypes from 'prop-types';

Below example shows it.

Parent Component

import ChildPropTypes from './ChildPropTypes';

const ParentPropTypes = () =>{

    return(

        <div>

            <ChildPropTypes name="Test-string"/>

        </div>

    );

}

export default ParentPropTypes;

Child Component

import PropTypes from 'prop-types';

const ChildPropTypes = (props) =>{

    return (

        <div>

            <p>{props.name}</p>

        </div>

    );

}

ChildPropTypes.defaultProps = {

    name: "name"

}

ChildPropTypes.propTypes = {

    name: PropTypes.string,

}

export default ChildPropTypes;

This will not given any warnings.

If we try to send number instead of string it will through warning shown below.

import ChildPropTypes from './ChildPropTypes';

const ParentPropTypes = () =>{

    return(

        <div>

            <ChildPropTypes name={1}/>

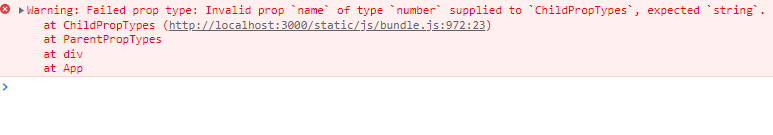
        </div>

    );

}

export default ParentPropTypes;

Warning shows below.



Child component prop expected string but parent try to send number for name property.

Follow the link for more details(<https://blog.logrocket.com/validate-react-props-proptypes/>).

**Hooks in React**

* Hooks allow function components to have access to state and other React features.
* Hooks allow us to "hook" into React features such as state and lifecycle methods.
* It does not work inside classes.

**When we use Hooks**

* If you write a function component, and then you want to add some state to it, previously you do this by converting it to a class. But, now you can do it by using a Hook inside the existing function component.

**Rules of Hooks**

* Hooks can only be called inside React function components.
* Hooks can only be called at the top level of a component.
* Hooks cannot be conditional.

1. **useState Hook**

The React useState Hook allows us to track state in a function component.

//need to import useState

import { useState } from "react";

const About = () =>{

    //initialize current state

    const [name, setName] = useState('Bhushan');

//update the functional component by using set state action

    const handleName = () =>{

        setName(name === 'Bhushan'?'Kumar':'Bhushan');

    }

    return (

        <div className="square border">

            <h1>Using useState Hook</h1>

            <div className="d-flex flex-row mb-3">

                <div className="p-2"><h1>About</h1></div>

                <div className="p-2"><h1>{name}</h1></div>

//calling the handleName to set the new state

                <div className="p-2"><button onClick={handleName}>Change Name</button></div>

            </div>

        </div>

    );

}

export default About;

1. **useEffect(**[**https://dmitripavlutin.com/react-useeffect-explanation/**](https://dmitripavlutin.com/react-useeffect-explanation/)**)**

* useEffect is a tool that lets us interact with the outside world but not affect the rendering or performance of the component.
* If the DOM is not ready state, if we try to send the API request or some time consuming actions it may effect the performance of the application, So use useEffect() hook to wait till DOM is ready. Once DOM is ready it well send the request and fetch the data. Follow below link(<https://www.youtube.com/watch?v=4xBLyslzDM0>)
* import { useEffect, useState } from "react";
* const UseEffectExample = () => {
* const [name, setName] = useState("initial Name");
* const [count, setCount] = useState(0);
* const [stopWatch, setStopWatch] = useState(20);
* //It will execute on every state changes.
* useEffect(() => {
* console.log(count + " Every re-render");
* });
* //It will execute only at first render.
* useEffect(() => {
* console.log(count + " Initial re-render");
* }, []);
* //It will execute only state of stopwatch modified.
* useEffect(() => {
* console.log(stopWatch + " On state of stop watch change");
* }, [stopWatch]);
* const updateStopWatch = () =>{
* setStopWatch(stopWatch-1);
* };
* const updateCount = () =>{
* setCount(count+1);
* };
* return(
* <div>
* <h1>Welcome</h1>
* <button onClick={updateStopWatch}>Decrease</button>
* <button onClick={updateCount}>Increase</button>
* </div>
* );
* }
* export default UseEffectExample;

1. **useContext**

* React Context is a way to manage state globally.
* It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.
* To do this without Context, we will need to pass the state as "props" through each nested component. This is called "prop drilling".
* Refer this Link to more (<https://www.w3schools.com/react/react_usecontext.asp>)

Ex:

**Parent component**

* import { createContext, useState, useContext  } from "react"
* import ComponentTwo from './ChildContextTwo'
* //Create a context using createContext as shown below
* export const context = createContext();
* const ParentContext = () =>{
* const [user, setUser] = useState("Bhush");
* return(
* // using .Provider we are passing the state by globally declared context. this state is
* //available to child components
* <context.Provider value={user}>
* <h1>{`Hello ${user}!`}</h1>
* <ComponentTwo />
* </context.Provider>
* );
* }
* export default ParentContext;

**Child component**

Here we can use the globally defined state as shown below. By using useContext hook we can maintain state globally, it will eliminate the passing data using props.

* import { useContext } from "react";
* import { context } from "./ParentContext";
* const ChildContextTwo = () =>{
* //using useContext hook we can get the globally declared state.
* const user = useContext(context);
* return(
* <div>
* <div>Context is needed in Context number 2 : {user}</div>
* <div></div>
* </div>
* );
* }
* export default ChildContextTwo;

1. **useRef**

* The useRef returns a mutable ref object.
* This object has a property called .current.
* The value is persisted in the refContainer.current property.
* These values are accessed from the current property of the returned object.
* The .current property could be initialised to the passed argument initialValue e.g. useRef(initialValue).
* The object can persist a value for a full lifetime of the component.
* For more (<https://www.geeksforgeeks.org/react-js-useref-hook>).

Below code is updated

import React, { useEffect, useRef, useState } from "react";

const UseRefExample = () => {

  const [count, setCount] = useState(0);

  const counter = useRef(0);

  let val = 0;

  // Since the ref value is updated in the render phase,

  // the value can be incremented more than once

  useEffect(() => {

    console.log("Rendered for count");

  });

  // useEffect(() => {

  //   console.log("Rendered for counter");

  // });

  // counter.current = counter.current + 1;

  //From following example we can use useRef for eliminating the re-renderings.

  //We just want the updated values, not for updating in UI then go for useRef().

  //useRef() available till end of the life of the hook.

  const handleClick = () =>{

    setCount(count + 1);

    console.log("Count is " + count);

  }

  const handleCount = () =>{

    counter.current = counter.current + 1;

    console.log("Current is " + counter.current);

  }

  const handleLocalLet = () =>{

    val = val + 1;

    console.log("val is " +val);

  }

  const handleCurrentValues = () =>{

    console.log("Count is(State): "+ count);

    console.log("Counter is(Ref): "+ counter.current);

    console.log("val is(Local): " +val);

  }

  return (

    <>

      <h1>{`The component has been re-rendered ${counter.current} and ${val} times`}

      </h1>

      <button onClick={handleClick}>Increase(State)</button>

      <button onClick={handleCount}>Counter(Ref)</button>

      <button onClick={handleLocalLet}>Counter(Let)</button>

      <button onClick={handleCurrentValues}>Current Values</button>

    </>

  );

};

export default UseRefExample;

1. **useReducer**

* useReducer is a hook user to state management.
* Alternative to useState Hook.
* useState is built using useReducer, useState is a primitive type.
* It takes two values and returns new state with pair of values.
* Follow the link (<https://www.youtube.com/watch?v=cVYp4u1m6iA>)
* Follow the link (<https://dmitripavlutin.com/react-usereducer/>)
* import React, { useReducer } from "react";
* const initialMessage = "Welcome";
* const reducer = (state, action) => {
* switch(action){
* case 'CEO':
* return state + " Test\_CEO\_Name";
* case 'Manager':
* return state + " Test\_Manager<Name";
* case 'TL':
* return state + " Test\_TL\_Name";
* default:
* return state + " User";
* }
* }
* const useReducerExample = () =>{
* //state always keeps till component exist
* const [message, dispatch] = useReducer(reducer, initialMessage);
* return(
* <div>
* <h1>{message}</h1>
* <button onClick={() => dispatch('CEO')}>CEO</button>
* <button onClick={() => dispatch('Manager')}>Manager</button>
* <button onClick={() => dispatch('TL')}>Team Lead</button>
* </div>
* );
* }
* export default useReducerExample;

Result: Welcome Test\_CEO\_Name Test\_Manager<Name Test\_TL\_Name Test\_CEO\_Name

It will keeps appending a string while you select any of the button.

1. **useCallback and useMemo**
2. **useCallback**

Before explaining useCallback(), let’s understand javascript functions, how to call the functions and how they create different objects. Code below shows.

function sumFunctionFactory() {

return (a, b) => a + b;

}

const function1 = sumFunctionFactory();

const function2 = sumFunctionFactory();

function1(2, 3);

// expected output: 5

function2(2, 3);

// expected output: 5

console.log(function1 === function2);

// expected output: false

* If we are call the same function multiple times it will create new object every time and it references the different instances.
* when a component re-renders, every function inside of the component is recreated and therefore these functions’ references change between renders.
* So by using useCallback() we can reuse the created objects again and again.
* useCallback(callback, dependencies) will return a memoized instance of the callback that only changes if one of the dependencies has changed. This means that instead of recreating the function object on every re-render, we can use the same function object between renders.

const memoized = useCallback(() => {

// the callback function to be memoized

},

// dependencies array

[]);

* The React useCallback Hook returns a memoized callback function.
* useCallback is a hook that will return a memoized version of the callback function that only changes if one of the dependencies has changed. Memorization is a way to cache a result.
* Follow the Link for more details(<https://www.youtube.com/watch?v=IL82CzlaCys>)

1. **useMemo**

* The useMemo Hook only runs when one of its dependencies update.
* The useMemo Hook can be used to keep expensive, resource intensive functions from needlessly running.
* Follow the link for more details(<https://www.youtube.com/watch?v=qySZIzZvZOY>)
* import React, { useMemo, useState } from "react";
* function Counter(){
* const [counterOne, setCountOne] = useState(0);
* const [counterTwo, setCountTwo] = useState(0);
* const incrementOne = () => {
* setCountOne(counterOne + 1);
* console.log("Counter one");
* }
* const incrementTwo = () => {
* setCountTwo(counterTwo + 1);
* console.log("Counter Two");
* }
* /\*\* using normal function it cause delay\*/
* function isEven(){
* let i = 0;
* while( i < 2000000000) i++
* console.log("Is Even");
* return counterOne % 2 === 0;
* }
* /\*\*\*\* It will create a delay for both the methods(incrementTwo is need to wait for incrementOne) \*\*\*\*/
* // const isEven = useMemo(()=>{
* //     let i = 0;
* //     while( i < 2000000000) i++
* //     return counterOne % 2 === 0;
* // });
* /\*\*\*\* We can resolve it's dependency delay by using useMemo \*\*\*\*/
* // const isEven = useMemo(()=>{
* //     let i = 0;
* //     while( i < 2000000000) i++
* //     return counterOne % 2 === 0;
* // }, [counterOne]);
* return(
* <div>
* <h1>Counter</h1>
* <div>
* <button onClick={incrementOne}>Count - {counterOne}</button>
* <span>{isEven() ? 'Even': 'odd'}</span>
* </div>
* <div>
* <button onClick={incrementTwo}>Count - {counterTwo}</button>
* </div>
* </div>
* );
* }
* export default Counter;